# Are Web Applications Ready for Parallelism?

Cosmin Radoi
University of Illinois, USA
cos@illinois.edu

Stephan Herhut
Intel Corporation, USA
stephan@herhut.eu

Jaswanth Sreeram
Intel Corporation, USA
jaswanth.sreeram@intel.com

Danny Dig
Oregon State University, USA
digd@eecs.oregonstate.edu

## Abstract

In recent years, web applications have become pervasive. Their backbone is JavaScript, the only programming language supported by all major web browsers. Most browsers run on desktop or mobile devices with parallel hardware. However, JavaScript is by design sequential, and current web applications make little use of hardware parallelism. Are web applications ready to exploit parallel hardware?

We answer the question in two steps: First, we survey 174 web developers about the potential and challenges of using parallelism. Then, we study the performance and computation shape of a set of web applications that are representative for the emerging web.

Our findings indicate that emerging web applications do have latent data parallelism, and JavaScript developers' programming style is not a significant impediment to exploiting this parallelism.

***Categories and Subject Descriptors*** D.1.3 [*Programming techniques*]: Concurrent Programming; D.2.7 [*Software engineering*]: Distribution, Maintenance, and Enhancement

***Keywords*** javascript, web, survey, performance

## 1. Introduction

Parallel hardware has become a reality of modern computing and its use is no longer confined to high performance applications and super computing. Even mobile phones now regularly feature multi-core CPUs and programmable GPUs. SIMD (Single Instruction, Multiple Data) extensions add further to the mix of exploitable hardware parallelism. Creating the best possible experience on any device therefore requires tapping into parallel hardware's potential to increase performance, save energy, or even both.

Most traditional platforms and languages have developed tools and language extensions to help developers adapt their code to run on modern parallel hardware. Yet, HTML5, an emerging web-based application ecosystem that promises portability across devices and form factors, and its implementation language JavaScript, seem still to be stuck in the sequential past. While browser vendors have invested heavily into the sequential performance of their JavaScript engines and added some support for concurrency [1], support for parallelism is still in its infancy. Parallel JavaScript [5] and WebCL [2] are two proposals to extend JavaScript to support parallel programming but neither is widely used. While this can be attributed to their prototypical implementation, the question remains: *Are web applications ready for parallelism?*

*PPoPP'15*, , February 7–11, 2015, San Francisco, CA, USA.
Copyright © 2015 ACM 978-1-4503-3205-7/15/02…$15.00.
http://dx.doi.org/10.1145/2688500.2700995

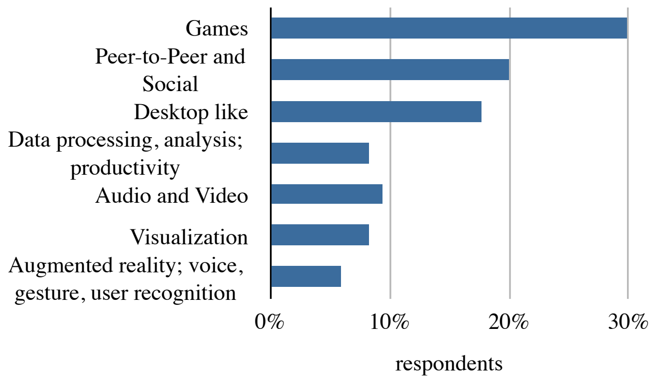

**Figure 1.** Categories of future web applications, identified by respondents.

Earlier work by Fortuna et al. [4] has found that typical web applications have potential for achieving significant speedup from concurrent execution. This is encouraging but most of the potential they found stems from independent tasks rather than loops. Therefore it would be hard to exploit it using massively data-parallel hardware like GPUs or SIMD. Even more, Richards et al. [6] have studied the runtime behavior of typical JavaScript applications and found widespread use of dynamic language features, which hinder execution on restricted hardware like GPUs and SIMD units. Both findings suggest that, while there is some potential for task parallelism, the web is not a fertile ground for data parallel programming.

While this conclusion might be true for the web of the past, our hypothesis is that it does not apply to the *emerging* web of applications. With the shift of the web to an era of application centric usages like, for example, image editing, augmented reality applications, and sophisticated gaming, the characteristics of executed code change, too. As these usages are more compute-intensive, they also are more likely to gain from data-parallel compute capabilities. Even more, due to the increased focus on application logic over just rendering content, we also expect other high-level code properties, like use of dynamic language features, to change. Lastly, a new generation of programmers might also bring different programming styles to the table, e.g., due to influences from more declarative programming patterns during their education.

Of course, measuring such a trend in its early phases is difficult. Most production-quality web sites are still built in a legacy style and new applications are only beginning to emerge. Analyzing currently-popular web sites would bias our results towards what works well on most platforms now, not the workloads that are missing precisely because they would require more performance. Thus, in contrast to earlier studies, we had no adequate top-100 list or similar to draw from. Instead, we chose to measure the change where it starts: with the shift in developers' opinion.

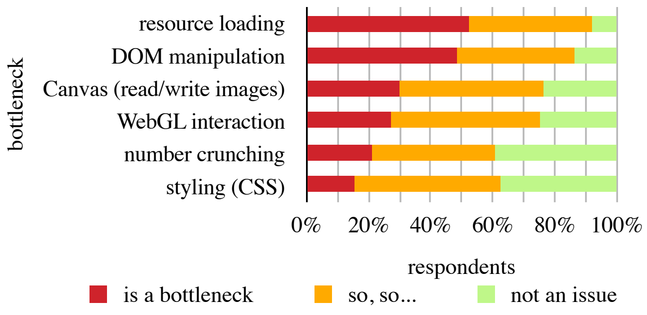

**Figure 2.** The importance of various performance bottlenecks, as scaled by respondents.

## 2. JavaScript in Practice: A Survey

We formulated a questionnaire consisting of 20 questions. The questions broadly fall into four categories: trends in web applications, programming style, preferred tools and frameworks, and perceived performance bottlenecks. We publicized the questionnaire using social media, with help from a few influential developers in the JavaScript community. We received 174 answers. We present here a small subset of the survey results.

### 2.1 Future Trends in Web Applications

We asked developers: *"In your opinion, what new kinds of applications will trend on the web over the next 5 years?"* We hand-coded their answers using *qualitative thematic coding* [3]. Figure 1 shows the resulting application categories.

As a general trend, we found that applications formerly at home on the desktop are predicted to transition to the web. With the flattening of per-core performance improvement, desktop applications have become increasingly parallel in the past few years. We expect that their web counterparts will also need to be parallel in order to be competitive.

### 2.2 Performance Bottlenecks in Current Web Applications

With the increasing richness and functionality embedded into web applications, especially real-time interactive applications, understanding typical performance bottlenecks is an important consideration for developers as well as for engine implementors.

Our survey asked the respondents to categorize each of several components as ”not an issue”, ”so, so...”, or ”is a bottleneck”. The aggregated responses are shown in Figure 2.

Confirming the common complaint in the JavaScript community, 53% and 48% of respondents mentioned that resource loading and DOM manipulation (e.g., inserting or deleting elements), respectively, are bottlenecks. 21% of respondents consider that number crunching/math computation is a bottleneck. While the percent may seem low compared to the opinion on other operations, we see it as significant in the context of current popular web sites, which usually do not execute any computationally-intensive algorithms.

## 3. Case Study

The survey gave us a general idea of web developers' preferences, and of emerging trends. The case study brings insight into the programming style and issues prevalent in the computationally-intensive parts of emerging web applications.

We selected 12 workloads from the categories mentioned by the developers and analyzed them for latent data parallelism. In particular, we were interested in the presence of parallelizable loops and their approximate percentage of execution time. For each application, we inspect the top loop nests that, together, make up at least two thirds of the application's time spent in loops. Altogether, we inspect 22 loop nests across 12 subject web applications.

About three fourths of the inspected loop nests have some intrinsic parallelism, i.e. do not have dependencies that we think could not be broken. Also, in most cases, the trip count and granularity is high enough for some form of data-parallelism to be potentially useful. Still, exploiting this parallelism may not be easy. In many cases it would require a combination of code changes and browsers with efficient parallel data structures and concurrent DOM and Canvas implementations.

We also looked at further code properties which may influence parallelism. We found that JavaScript poses the traditional issues to parallelization, while also raising new ones that stem from its evolving, dynamic, and web-centric nature. In addition to discovering available parallelism and matching the parallel computation to the hardware, a JavaScript programmer also needs to get around concurrent updates to the non-concurrent DOM, concurrent reads and writes of global memory, and polymorphic variables.

Overall, our findings differ from earlier work and we found a surprisingly large quantity of compute-intensive loops of which many were latently parallel.

## 4. Conclusion

With the proliferation of desktop and especially mobile operating systems, the web is increasingly seen as a cross-platform solution for delivering applications. In our survey, when asked about emerging trends in web applications, JavaScript developers mostly identified kinds of applications that, not long ago, were only available as native desktop applications. But this transitioning to the web comes with a challenge: native desktop applications had to resort to multi and many-core parallelism for performance. Should the web follow suit? If so, how hard will it be?

To answer these questions we conducted a survey among JavaScript developers asking them about their use of JavaScript language-features that may impede parallelism. Furthermore, we did a case study looking at the computationally-intensive loop nests in 12 web applications. While JavaScript is highly dynamic, we found that developers seldom use language features that impede parallelism. An important current limitation is that browsers have non-concurrent implementations of basic data structures (e.g., the DOM). Much of the compute-intensive code we inspected is written in a style typical of non-dynamic imperative languages. This means that many of the lessons learned by the programming community while parallelizing desktop applications will translate to the web.

A detailed technical report of our findings is available at:
`http://hdl.handle.net/2142/72643`

## Acknowledgment

This research is partly funded through NSFCCF-1439957 and CCF-1442157 grants, a SEIF award from Microsoft, and a gift grant from Intel.

## References

[1] Web Workers. http://www.w3.org/TR/workers/.

[2] webCL. http://www.khronos.org/webcl/.

[3] D. S. Cruzes and T. Dyba. Recommended steps for thematic synthesis in software engineering. In *ESEM '11*. IEEE.

[4] E. Fortuna, O. Anderson, L. Ceze, and S. Eggers. A limit study of javascript parallelism. In *IISWC '10*.

[5] S. Herhut, R. L. Hudson, T. Shpeisman, and J. Sreeram. River Trail: A path to parallelism in JavaScript. In *OOPSLA '13*.

[6] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. In *PLDI '10*.